INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

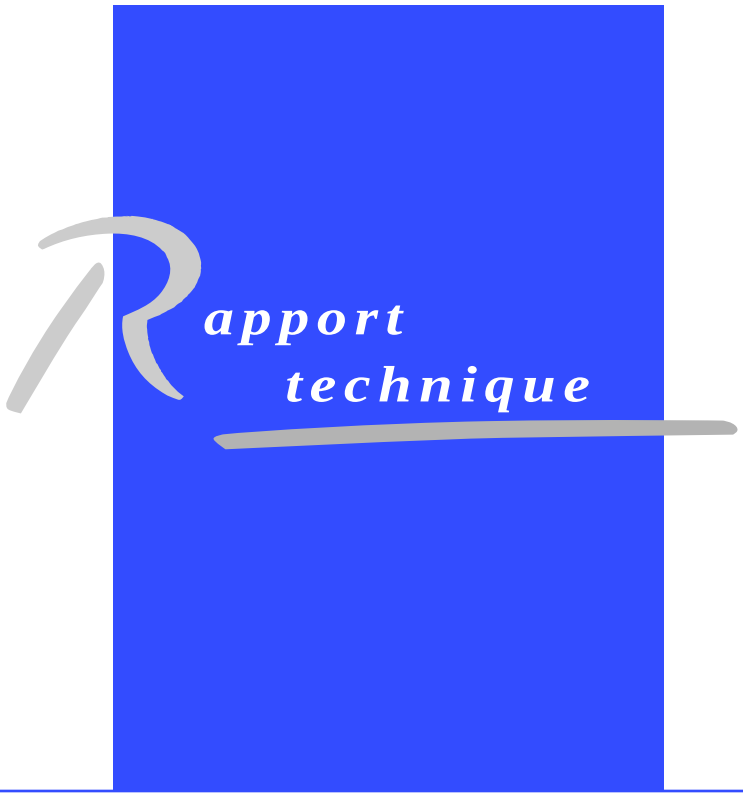
Approche orientée composant d'une pile réseau

Yvan Royon — Stéphane Frénot — Antoine Fraboulet

N° 0298

Aout 2004

THÈME 1



ISSN 0249-0803

Approche orientée composant d'une pile réseau

Yvan Royon[*] , Stéphane Frénot[†] , Antoine Fraboulet[‡]

Thème 1 —Réseaux et systèmes
Projet ARÈS

Rapport technique n° 0298 —Aout 2004 —29 pages

Résumé : La plupart des applications actuelles sont dotées de capacité de communication via un réseau, typiquement Internet. Cette communication se base sur un service rendu par la pile réseau du système d'exploitation.
Nous proposons Mynus, une pile réseau basée sur un modèle à composants. Ses objectifs sont d'apporter les propriétés de reconfigurabilité, d'adaptabilité et de résilience aux pannes, tant au niveau des empilements protocolaires que des protocoles eux-mêmes.
De plus, si les communications entre les différents composant de la pile ont un coût, nous pensons qu'il est possible de le compenser. Par son architecture, Mynus permet de factoriser certains services, qui sont rendus actuellement à plusieurs niveaux : une ou plusieurs couches protocolaires, ainsi que divers middlewares.

Mots-clés : Pile réseau, Composants, Intergiciel

[*] yvan.royon@insa-lyon.fr
[†] stephane.frenot@insa-lyon.fr
[‡] antoine.fraboulet@insa-lyon.fr

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 —Télécopie +33 4 76 61 52 52

Component-based Approach for a Network Stack

Abstract: Most of the current application softwares have the ability to communicate via a network, usually the internet. Such communications are based upon a service provided by the Operating System's Network Stack.
Our proposition, called Mynus, is a Network Stack based on a component model. Its goals are to provide reconfigurability, adaptability and fault-tolerance at two levels: the suite of protocols that define the Network Stack, and the protocols' inwards.
Besides, although the communications between the Network Stack's components do have a cost, we believe it can be compensated. By design, Mynus allows to regroup services that are often provided at more than one layers - protocol or middleware.

Key-words: Network Stack, Components, Middleware

1 Introduction

Depuis une quinzaine d'années, dans le monde de la conception des applications et des systèmes, deux grands types de *design* se font concurrence : l'approche monolithique et l'approche modulaire. Il est admis qu'un design monolithique donne de meilleures performances brutes. Une approche modulaire permet, elle, de rajouter des fonctionnalités plus facilement et plus "proprement", tout en étant plus tolérant aux pannes.

Nous nous attachons ici à concevoir la pile réseau d'un système, sous forme de composants. Nous entendons par pile réseau les protocoles usuels correspondant aux niveau OSI 2 à 6 (Ethernet, ARP, IP, ICMP, UDP…), ainsi que les mécanismes permettant aux applications d'utiliser ces protocoles. Cette pile réseau est conçue suivant une architecture "système", qui définit les éléments en présence (structures de données, traitements) ainsi que leurs communications (échanges de messages). Nous proposons d'utiliser une architecture à base de composants, plutôt qu'une conception monolithique telle que celle des systèmes d'exploitation usuels (*e.g.* la pile BSD).

Les avantages d'une pile réseau à base de composants peuvent être multiples :

- Couche protocolaire reconfigurable à chaud, de manière transparente à l'application. Par exemple, substituer un protocole par un autre moins gourmand (*e.g.* TCP par UDP), ajouter ou enlever un intermédiaire (*e.g.* utiliser IPSec sur IP), *etc.*
- Reprise transparente sur erreur. En cas de "plantage" d'un composant, on veut pouvoir le recharger avec un impact minimum sur l'application.

Ce mémoire débute par un rappel des besoins des applications qui utilisent le réseau pour communiquer. Nous faisons un tour d'horizon des technologies existantes permettant de réaliser ces communications. Ceci est suivi d'un état de l'art de la recherche dans les approches orientées composant. Nous comparons, dans divers domaines, ces approches orientées composant avec les approches existantes. Nous exposons ensuite l'architecture de notre pile réseau, baptisée $\mathcal{M}ynus$, pour enfin discuter de ses forces et faiblesses.

2 Contexte : mise en œuvre d'applications communiquantes

De très nombreuses applications aujourd'hui peuvent ou doivent communiquer avec des applications distantes. Cela va du logiciel de travail collaboratif au logiciel Microsoft qui envoie son numéro de série aux serveurs de la société, en passant par les logiciels qui vérifient leurs mises à jour. Pour cela, ces applications doivent disposer d'un point d'entrée sur la pile réseau utilisée sur Internet (typiquement TCP/IP). Les applications voient ce point d'entrée comme un service rendu par le système d'exploitation. Voici une liste non exhaustive de tels services.

2.1 API niveau "système"

2.1.1 Socket & TLI

Les Application Programming Interfaces (API) les plus courantes, qui forment la brique de base d'accès au réseau, sont les Sockets (initialement de BSD) et Transport Layer Interface (de AT&T). Ce sont des abstraction de TCP/IP, qui peuvent être vues comme des tunnels où transitent des paquets d'octets.

- Avantages :
  - l'API est simple d'utilisation ;
  - elle est fournie par la plupart des systèmes ;
  - le code C est quasi-portable (*e.g.* entre Unix et MS Windows).
- Inconvénients :
  - les fonctionnalités sont rudimentaires (connecter, envoyer un paquet, recevoir un paquet, déconnecter).

2.1.2 RPC, RMI & CORBA

CORBA et RMI sont des surcouches aux sockets qui fournissent des services supplémentaires. En l'occurence, ils permettent l'accès à des commandes applicatives ou des objets Java distants. Ces deux couches utilisent les Remote Procedure Calls pour la partie transport.

- Avantages :
  - notion de stub : un élément local se "fait passer" pour un élément qui est en fait distant, en masquant le déport de code ;
  - notions de sérialisation et de marshalling : transférer des données typées ou des objets via un flux d'octets ;
  - notion de registre : un annuaire recense les objets servis. Ceci permet à des clients potentiels de l'objet d'y accéder sans connaitre au préalable le serveur ;
  - les interfaces et leur implémentation sont séparées.
- Inconvénients :
  - le service n'est pas garanti : il dépend de la disponibilité du réseau ;

- les limites sont celles de la notion d'objet : pas de reconfiguration, pas d'administration, peu évolutif…

2.1.3 SOAP

Pour répondre aux contraintes d'extensibilité, et pour s'affranchir des problèmes de structuration des données, SOAP propose l'envoi unidirectionnel de messages XML.

– Avantages :
  - l'idée de base est simple ;
  - par nature, le langage XML est extensible ;
  - le XML peut être encapsulé par TCP, HTTP, SMTP…

– Inconvénients :
  - le service n'est pas garanti : il dépend de la disponibilité du réseau et du serveur ;
  - par nature, le langage XML est lourd et volumineux ;

2.2 middlewares réseau

2.2.1 EJB / CCM / .NET

Les Enterprise JavaBeans, Corba Component Model et .NET sont respectivement proposés par Sun, l'Object Management Group et Microsoft. Leur objectif est de fournir des mécanismes de déport de composants dans le cadre d'applications distribuées. Il s'agit de composants *middleware* et applicatifs, et non au niveau du système d'exploitation.

– Avantages :
  - la notion de composant permet de séparer les aspects fonctionnels et non fonctionnels. Le *middleware* lui-même prend en charge certains aspects non fonctionnels, *e.g.* la distribution ou la tolérance aux pannes ;
  - elle permet également d'en contrôler le cycle de vie ;
  - on voit apparaître la problématique de déploiement des composants, à l'aide de descripteurs (XML ou autres).

– Inconvénients :
  - ces *middlewares* sont explicites, dans le sens où ils imposent de grandes contraintes de programmation
  - l'extension et l'adaptation des composants ne sont quasiment pas gérés par le framework.

2.2.2 MoM

Les Message-oriented Middlewares permettent à des applications de s'échanger des messages de manière asynchrone, via un tiers (une boîte aux lettres). On peut citer en exemple l'API de Sun, JMS (Java Message Service) [15].

– Avantages :
  - la communication est asynchrone ;
  - le couplage des applications est faible, et le format des messages est indépendant du langage et de l'architecture ;
  - la livraison des messages est garantie.
– Inconvénients :
  - il n'existe pas de norme prépondérante en-dehors de JMS.

2.3 Besoins des nouvelles applications réseau

Les applications d'aujourd'hui nécessitent de plus en plus souvent de mettre en œuvre des services "avancés" : haute disponibilité, mobilité, gestion de la qualité de service (QoS), possibilités de mise à jour, utilisation d'un réseau pair à pair…
Pour réaliser ces services avancés, des *middlewares* dédiés sont conçus et développés. D'un point de vue conceptuel, il n'est pas satisfaisant de voir se multiplier les couches intermédiaires, ni de constater une certaine redondance dans les services rendus : routage IP et routage pair à pair, correction d'erreurs aux niveaux physique, transport et application, fragmentation et fenêtres d'émission à plusieurs niveaux, *etc*.

Partant de ces constats, nous proposons d'utiliser une approche à composants pour définir une pile réseau. Celle-ci doit être reconfigurable, modulable, adaptable suivant les cas d'utilisation (*e.g.* appareils mobiles…). L'objectif est de "descendre" des services offerts par les *middlewares* au niveau du système, afin d'éviter les redondances et la dispersion des services.
L'approche naturellement pressentie pour une telle pile réseau est une approche orientée composant.

3 État de l'art

Nous avons vu divers services de communication offerts aux applications. Tous sont basés sur les API de type Socket, et proposent des services supplémentaires permettant aux applications de faire abstraction du réseau.
Voyons à présent l'état de l'art de la recherche, en matière de modèles à composants. Nous comparons ensuite l'application de ces modèles à des cas spécifiques (systèmes d'exploitation, piles réseau, *middlewares*) avec leurs équivalents existants, qui suivent un modèle classique monolithique.

3.1 Approches orientées composant

3.1.1 Historique

Au commencement de la programmation "moderne" était l'approche procédurale. Son langage par excellence, le C, permit de créer des applications avec de bonnes performances brutes. En effet, cet aspect "bas niveau" servit à réaliser des systèmes d'exploitation de renom, tels les Unix.
Puis vint l'approche orientée objet. Cette approche proposa un typage fort des données, ainsi qu'une vision plus fonctionnelle du code. Son langage emblématique, le Java, popularisa dans le monde du système la notion de machine virtuelle. Quoi que fussent les défauts de cette dernière (taille mémoire, lenteur d'exécution), elle s'imposa car fournissant un environnement portable.
Aujourd'hui, l'heure est à la rationalisation de la conception et du code, à la réutilisation, à l'exploitation des "bonnes pratiques". En particulier, on se penche de plus en plus sur l'approche orientée composant. Le terme "composant" a une signification différente pour chaque personne qui l'emploie ; nous allons donc faire un tour d'horizon des modèles, des mises en œuvre système et des mises en œuvre applicatives basées sur des composants.

3.1.2 Modèles

Il existe dans la littérature plusieurs modèles à composants. Ces modèles définissent chacun leur sémantique : ce qu'est un composant, comment il communique, comment son *framework* le gère. Nous voyons ici les principaux modèles et les concepts qui les accompagnent.

- **Fractal**

  Fractal [5] [17] d'ObjectWeb est un modèle à composants, pour la conception et l'implantation d'applications et systèmes reconfigurables. Sa communauté utilisatrice est principalement centrée autour du *middleware*, où les problèmes de reconfiguration et de déploiement sont cruciaux.

  Pour Fractal, comme pour les autres modèles cités, un composant est en première approximation un objet, isolé du monde extérieur par des interfaces. On distingue ici les interfaces serveur (les services rendus par le composant), les interfaces client (les services requis) et les interfaces de gestion & configuration. Une interface client et une interface serveur peuvent communiquer lorsqu'elles sont liées (opération *binding*). Pour se faire, chaque interface est identifiée par un nom, et la recherche d'interfaces est déléguée à un service d'annuaire.

Le composant est séparé en une partie fonctionnelle (le contenu) et une partie non fonctionnelle (le contrôleur). Le contrôleur peut être manipulé par le biais des interfaces de configuration. Il dispose également d'intercepteurs, pour relayer les interfaces fonctionnelles vers le contenu du composant. Le contrôleur est donc un passage obligé entre le contenu du composant et le message extérieur.
Le modèle Fractal s'attache tout particulièrement à l'architecture de l'application. Ainsi, deux types de composants sont définis : les composants de base (ou élémentaires), qui sont les briques fonctionnelles de l'application, et les composants composites, qui regroupent et architecturent des composants de base. Pour garantir l'interopérabilité de composants écrits dans des langages de programmation différents, Fractal donne la possibilité d'utiliser un langage de définition d'interface (IDL).
Le modèle propose un ensemble d'interfaces de configuration, permettant de gérer :
– *Introspection* : pour découvrir les interfaces externes fournies par le composant ;
– *Liaison* : pour lier les interfaces client du composant ;
– *Configuration* : dans le cas d'un composite, pour ajouter et supprimer des sous-composants ;
– *Cycle de vie* : pour interrompre et démarrer le composant, notamment si l'on veut changer sa configuration ou ses liaisons en cours d'exécution.

Chacune de ces interfaces, tout comme l'IDL, est optionnelle. Sans aucune de ces interfaces, un composant est simplement un objet proprement isolé. Ajouter ces interfaces permet de gagner des propriétés intéressantes, mais au prix d'une baisse de performances : c'est un compromis à faire lors de la conception.

Julia est une implantation en Java de la spécification Fractal. Il s'agit d'un *framework* initialement prévu pour développer la partie contrôle des composants. Pour cela, Julia propose un ensemble de *mixins*. Ce sont des classes abstraites, qui implantent des fonctionnalités réutilisables. Les classes implantant la partie contrôle du composant sont obtenues en intégrant un groupe de *mixins* au sein d'un squelette.

- **Avalon**

  Le projet Apache Avalon [9] est principalement orienté vers les applications serveurs. Tout comme Fractal, il propose la création de composants et la gestion de leur cycle de vie. Par contre, il définit précisément l'annuaire de composants (le ServiceManager), alors que Fractal laisse cette tâche au développeur.

  Un composant qui fournit des services enregistre ses "rôles" auprès du ServiceManager. Puisque Avalon cible les applications serveurs, il définit pour les composants "serveurs" trois modes de fonctionnement :
  – *Single threaded* : sert un seul client ;
  – *Thread safe* : sert plusieurs clients ;
  – *Poolable* : sert un client, mais est réutilisable.

  Avalon est complété par plusieurs sous-projets. En particulier, Merlin est un conteneur qui facilite la gestion, l'assemblage et le déploiement dans les applications orientées composant.

- **OSGi**

OSGi [2] se place dans le cadre du délivrement de services, par un fournisseur et via un opérateur, directement au domicile du particulier. Les applications visées sont les équipements de la maison communicante (cuisine, électroménager...) ou encore le contenu multimédia à la demande (du type *pay-per-view*).

Les fournisseurs de service déploient des *bundles* sur la plate-forme OSGi côté client. La plate-forme est un conteneur, et les *bundles* sont des composants. En plus des composants donnés par les fournisseurs, la plate-forme héberge des services locaux. Le conteneur gère les dépendances entre *bundles* et les liaisons de *bundle* à service ; il gère également le déploiement et l'administration des composants.

Comparé à Fractal, OSGi ne propose pas de composition hiérarchique, et n'intègre pas les aspects non fonctionnels des composants.

3.2 Mises en œuvre

Voyons à présent l'application de modèles orientés composants aux cas concrets qui nous intéressent : les systèmes d'exploitation en général, et les piles réseau en particulier. Nous pouvons ainsi comparer les travaux réalisés avec leurs penchants monolithiques déjà utilisés.

3.2.1 Systèmes d'exploitation

- **Approche classique**

  Les systèmes d'exploitation usuels, *e.g.* les Unix ou MS Windows, sont développés majoritairement en langage C, de manière procédurale et monolithique. L' avantage est (normalement) d'avoir de bonnes performances en terme de rapidité d'exécution, puisque le langage utilisé est proche du langage machine. Les inconvénients sont que le code est relativement difficile à relire et à faire évoluer, car il est volumineux.

  Les systèmes à micro-noyau sont, par définition, plus compacts, donc probablement plus simples à prendre en main. Un micro-noyau ne regroupe que les fonctionnalités les plus essentielles (*e.g.* gestion basique de la mémoire, gestion des communications, ordonnancement et gestion des threads). Tout le reste, *i.e.* le système de fichiers, la pagination de la mémoire ou encore les pilotes de périphériques, est lancé sous forme de services. L'intérêt est d'avoir un système plus robuste car résilient aux pannes. De plus, on obtient une vision plus fonctionnelle du système, car les différents services sont bien isolés. En revanche, les systèmes à micro-noyau subissent une perte de performances [13], principalement dues aux changements de contexte et aux communications entre les services.

  Du côté du monde Java, les machines virtuelles usuelles (Sun JVM, Blackdown, Kaffe...) suivent également un design monolithique. Elles sont codées en Java et en C. On trouve également des systèmes d'exploitation Java, tels JNode [20] (*Java New Operating System Design Effort*), où machine virtuelle et système d'exploitation ne font qu'un. JNode, écrit en Java et en assembleur, est également conçu de manière monolithique.

- **Approche orientée composant**
  Un système d'exploitation à composants fait référence à son design, et non à son implantation. En effet, il peut être implanté sous forme monolithique comme sous forme de micro-noyau.

  Think [7] est un *framework* pour les noyaux de systèmes d'exploitation, au design proche de Fractal (d'ailleurs créé par les mêmes équipes). L'architecture de ces systèmes d'exploitation n'est pas imposée par le *framework* : elle peut être monolithique, micro-noyau ou encore spécifique pour une application.
  Par rapport à Fractal, la notion de composite est abandonnée pour celle de domaine. Un domaine sert de conteneur aux composants de base, mais il n'est pas récursif. Le domaine est muni des interfaces de contrôle (*e.g.* cycle de vie), et les composants de base sont munis des interfaces fontionnelles. Kortex est une implantation de Think, dédiée aux processeurs Apple PowerMac (G3 et G4). Elle contient des éléments fonctionnels usuels aux systèmes d'exploitation : gestion de la mémoire, système de fichiers, périphériques d'entrée et d'affichage, réseau...Ces éléments peuvent alors être architecturés et étendus.

  Parmi les systèmes d'exploitation Java, JX [11] reprend la notion de domaine introduite par Think. Un domaine JX est un domaine d'exécution qui contient des classes Java, lesquelles font office de composants. Il s'agit donc d'une architecture hybride entre une approche orientée objet et une approche orientée composant.

3.2.2 Piles réseau

- **Modèle OSI**
  Le modèle OSI (Open Systems Interconnection) [19] de l'ISO définit les différents mécanismes à mettre en œuvre pour faire communiquer des applications via un réseau. Ces mécanismes sont répartis en 7 couches distinctes :

  1. *Physique* : le lien physique où transitent les trains de bits. Comprend le codage et la modulation du signal.
  2. *Liaison de données* : l'échange de trames d'octets au niveau du réseau local.
  3. *Réseau* : le routage et le service d'adressage lorsque plusieurs réseaux sont interconnectés.
  4. *Transport* : la frontière entre couches basses (le transfert de données) et couches hautes (les échanges applicatifs). Gère le transfert de données de bout en bout.
  5. *Session* : synchronisation des échanges pour une connexion.
  6. *Présentation* : mise en forme des données.
  7. *Application* : l'application communiquante.

  En pratique, les protocoles proposés par l'OSI sont rarement utilisés. On préfère par exemple les équivalents du modèle Internet : TCP ou UDP pour la couche 4, IP pour la couche 3, *etc*. La notion d'encapsulation des données de couche en couche reste cependant prépondérante

dans toutes les piles protocolaires utilisées. Cela force la vision "en couche" de ces piles : l'élément central est le protocole. Ce protocole traite des paquets d'octets (les encapsule), et les passe au protocole suivant. L'intelligence se situe donc au niveau des protocoles, tandis que les messages échangés ne sont que des structures de données.

Cette organisation figée des mécanismes réseau pose cependant certains problèmes. D'un point de vue conceptuel comme dans l'enseignement du réseau [22], la notion de couche masque celle de service. En effet, entre les modèles théoriques et l'évolution des modèles du marché, certains services (*e.g.* le contrôle de flux, la compression des données...) peuvent se retrouver dans plusieurs couches. De même, lorsque de nouveaux services apparaissent (*e.g.* les *middlewares*), le modèle est trop rigide pour pouvoir les intégrer.

- **Approche classique**

  L'implantation BSD de TCP/IP [23] est la référence en matière de piles réseau monolithiques. Elle sert de modèle pour les piles réseau des systèmes d'exploitation usuels. Elle regroupe en particulier des protocoles de niveau OSI 2 à 4, la couche Socket, et des structures de données (les *mbufs*) pour gérer les paquets réseau reçus et à émettre.
  Cette pile est reconnue pour être complète, optimisée "bas niveau" en performances brutes, le tout en 15 000 lignes de code en C (pour la version 4.4BSD-Lite). Par conséquent, bien que le code soit commenté, il n'est pas trivial de le reprendre pour l'améliorer ou l'étendre.

  La machine virtuelle Java de Sun [14] propose une API Java complète et fonctionnelle pour les sockets du domaine Internet. L'implantation de cette API fait appel à des méthodes natives JNI (Java Native Interface), qui elles-mêmes masquent des appels à l'implantation système (*e.g.* BSD) des sockets.

  JNode (*c.f.* 3.2.1) par contre dispose d'une partie réseau implantée en Java. Un paquet à émettre est un objet particulier, qui offre une structure de données et des méthodes de manipulation de base. Cet objet contient des en-têtes de niveau OSI 2, 3 et 4. L'implantation des protocoles réseau a pour but de remplir ces en-têtes, sans maintenir l'état du protocole (fenêtre de congestion, table de routage...) et sans souci de reconfiguration des protocoles. La pile elle-même est extrêmement statique (un protocole de niveau 4, un de niveau 3 et un de niveau 2), ce qui est à l'opposé de nos objectifs.

- **Piles réseau dans l'espace utilisateur**

  Des tentatives existent [4] pour implanter le protocole TCP dans l'espace utilisateur Unix. En l'occurence, la partie décapsulation du protocole est laissée dans l'espace noyau, pour des raisons de sécurité principalement, et tout le reste est passé dans l'espace utilisateur. Les auteurs arguent qu'il est plus facile ainsi de modifier, améliorer et débugger le protocole, et surtout que cela permet d'expérimenter des techniques d'implantation plus performantes. La limite de cette approche est que les performances sont très réduites par les nombreux appels système.

- **Approche orientée composant**

X-Kernel [12] est un *framework* permettant de décrire des protocoles réseau sous forme d'objets. Ces objets communiquent par des interfaces : une interface de *service* pour les autres protocoles de la même machine, et une interface dite *peer-to-peer* pour les protocoles équivalents sur les machines distantes. L'interface de *service* est standardisée, ce qui rend les protocoles interchangeables (*i.e.* la couche protocolaire est facilement modifiable). L'implantation X-Kernel des protocoles traduit les commandes de l'interface de *service* en commandes conformes RFC.
Un aspect à relever est l'existence d'un graphe des protocoles, qui donne les empilements protocolaires réalisables, ainsi que d'un graphe de "sessions", qui sont des instances d'un protocole dédiées à une communication.
Pour notre approche, X-Kernel n'est cependant pas suffisant : il lui manque la notion de reconfiguration des protocoles eux-mêmes. De plus, le graphe des protocoles est créé statiquement à l'initialisation du *framework*, alors que nous le voulons modifiable "à chaud".

3.2.3 Simulateurs

Certains éditeurs de logiciels proposent des simulateurs de réseau. Ils servent par exemple à faire des tests de performances sur de gros réseaux, sans avoir l'infrastructure matérielle à disposition.

- **OPNet Modeler**
  Ce logiciel [18], leader du marché, permet de modéliser un réseau des applications utilisées jusqu'aux protocoles de communication, en passant par les files d'attentes des équipements. OPNet dispose de "sondes", pour mesurer par exemple les débits atteints sur un lien donné. Pour calculer ces débits, le logiciel recrée les envois applicatifs, les encapsulations par chaque protocole, l'émission physique. Il dispose donc d'une modélisation relativement précise des protocoles comme IP ou TCP. **??**ependant, cette modélisation est réalisée dans un langage proto-C et dans des formalismes propres à OPNet. Elle n'est donc pas aisément réutilisable.
- **Omnet++**
  Omnet++ [6] est le penchant *open source* d'OPNet. Ici, un réseau est représenté par un ensemble de modules, qui s'échangent des messages. Contrairement à Opnet, Omnet++ n'offre pas de grain suffisant pour descendre au niveau des machines à état. Il n'est donc pas possible de modéliser un protocole réseau grâce au moteur d'Omnet.
  Un paquet, quel que soit son niveau OSI, est représenté par un champ de données, avec un attribut "protocol" positionné. La notion de protocole est donc réduite à un train de bits d'une longueur donnée. Il n'y a pas d'implémentation de TCP/IP.

  Il existe un ajout à Omnet++, nommé IPSuite, qui offre des implémentations en particulier d'IPv4, TCP et UDP. Ces implémentations C++ sont compilées à froid, puis utilisées avec le moteur d'Omnet compilé à froid. Les seules interfaces disponibles sur ces implémentations sont les interfaces de communication entre protocoles (les passages de PDU). Il n'existe pas d'interface de contrôle ou de configuration des couches protocolaires.

En conclusion, le simulateur OPNet Modeler est à priori intéressant pour ses modélisations de protocoles, mais se révèle inutilisable car trop figé dans l'architecture propriétaire du logiciel.

3.3 Middlewares de composition

Nous entendons par *middlewares* de composition des *frameworks* permettant de construire des applications, par opposition aux *middlewares* leur permettant de communiquer (*c.f.* 2.2).

3.3.1 Dream

Dream [16] est un *framework* pour les *middlewares* de communication, avec une préférence pour les MOM (*middlewares* orientés messages). L'objectif est de pouvoir construire, configurer et déployer ces *middlewares*, en s'appuyant sur le *framework* Fractal.

Dream utilise de manière intensive la notion de *mixin* (*c.f.* 3.1.2), pour maximiser la factorisation de code tout en mettant en avant leur côté fonctionnel. Il réutilise l'API Socket, en l'emballant dans des services de flux de données, offerts aux applications. Ce framework est donc une surcouche à la pile réseau. *Mynus* et Dream devraient donc être vus comme complémentaires : leur association permettrait de garder une approche orientée composant, de l'interface utilisateur au *driver* de la carte réseau.

3.3.2 Seda

L'objectif de Seda (Staged Event-Driven Architecture) [21] est de construire des applications serveurs supportant des milliers de connections concurrentes. Pour cela, cette architecture propose de décomposer une application en un ensemble d'*étapes*. Ces *étapes* communiquent via des files d'attente, donc de manière asynchrone. Ceci permet de faire du contrôle de charge, ainsi que d'intégrer les avantages des MOM (*c.f.* 2.2.2) au niveau des applications mêmes plutôt qu'au niveau de leurs communications avec d'autres applications seulement.

Notre objectif est de formaliser une pile réseau reconfigurable, adaptable et résistante aux pannes. Nous pouvons pour cela nous baser sur des modèles et implantations vus dans ce chapitre. En particulier, les modèles orientés composant, qu'ils soient génériques ou appliqués aux piles réseau, sont pour nous un élément clé.

4 Proposition

4.1 Modèle orienté composant

Avant de choisir un modèle orienté composant, définissons ce que doit être un composant pour répondre aux besoins de *Mynus*. Il s'agit d'un élément isolé du monde extérieur par ses interfaces. Celles-ci peuvent être des entrées/sorties (communication synchrone), des sources et puits d'événements (communication asynchrone, *c.f.* 2.2.2), des "prises" pour la (re)configuration, des facilités de déploiement. Nous faisons ici abstraction des interfaces de configuration "bas niveau" indispensables, telles que le lien au service d'annuaire ou la liaison entre composants.
Les modèles cités précédemment ne répondent pas parfaitement à chacune de ces contraintes. Nous allons cependant nous rapprocher du formalisme Fractal ; si la problématique de déploiement y est presque passée sous silence, le modèle est suffisament souple pour s'adapter au cas d'une pile réseau. De plus, Fractal est le seul à intégrer la notion de partage de composants. Enfin, ce modèle propose une API minimale, donc plus simple à appréhender.
Nous utilisons également plusieurs *design patterns* [10], pour asseoir la rationnalité de notre approche.

Dans ce chapitre, nous définissons tout d'abord ce qu'est un protocole d'un point de vue fonctionnel et générique. Nous explicitons ensuite les différents composants en présence, leur rôle et leur mode d'interaction. Ce dernier est illustré d'un cas d'utilisation concret, avant d'aborder certaines problématiques liées au système.

4.2 Approche fonctionnelle de la pile réseau

Pour remettre la problématique de service au centre de l'approche, nous isolons les tâches de base que peut accomplir un protocole sur un message reçu ou à envoyer sur le réseau. Chaque protocole masque alors ses traitements spécifiques par une interface générique :

- encapsuler : ajoute un en-tête et/ou une fin de message ;
- décapsuler : enlève en-tête et fin de message ;
- modifier : change le contenu du message ;
- fragmenter : éclate le message en plusieurs messages ;
- fusionner : réassemble des messages ;
- passer un message : envoyer le message à une autre entité, protocole ou application.

Un protocole est asynchrone ou synchrone, *i.e.* ses passages de message sont respectivement non bloquants ou bloquants en l'attente d'une réponse. Dans le cas d'un protocole asynchrone, nous choisissons un mode de communication par message ; dans le cas d'un protocole synchrone, le passage de message se fait par un appel direct sur le modèle IPC (Inter-Process Communication).

4.3 Mise en œuvre

4.3.1 Principe

Nous faisons le choix d'adopter un modèle de pile protocolaire proche de X-Kernel. Le fonctionnement de la pile est centré autour d'une entité "Message" (ici un composant), qui demande à être traité de protocole en protocole, suivant un graphe donné.
Nous estimons cependant que X-Kernel n'est pas allé suffisament loin dans cette approche, car les messages X-Kernel ne font que se transmettre de protocole en protocole. Cela revient à laisser les protocoles réseau se passer des "objets" au lieu de se passer une structure de données (comme dans les piles réseau classiques, *e.g.* BSD). Nous préférons faire des messages des entités d'exécution : les composants Message parcourent un graphe de protocoles (*c.f.* 4.3.4), et invoquent des traitements sur chaque élément (chaque protocole) de ce graphe. Il s'agit du *design pattern* Visitor.
La figure 1 présente les différents composants qui forment *Mynus*, lesquels sont explicités plus loin.

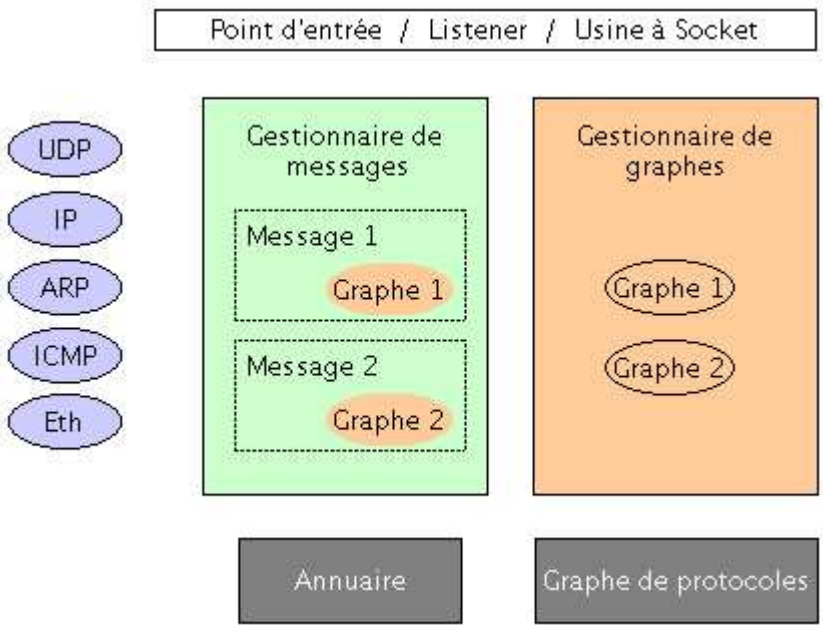


FIG. 1 – Organisation des composants du système

4.3.2 Composants Message

Un composant de type Message dispose d'une structure de données, représentant le message et les en-têtes à envoyer sur le réseau, et d'un état. Cet état représente la position du message dans le graphe de protocoles (*c.f.* 4.3.4) ainsi que sa direction (émission ou réception). Le graphe de protocoles est un composant partagé : le composant Message le voit comme un élément interne, alors qu'un graphe est un composant indépendant, potentiellement utilisé par plusieurs messages. Un composant Message importe un nombre très restreint d'interfaces :

- `getNextProto()` : pour obtenir le protocole suivant dans le graphe ;
- `visit()` : pour se faire traiter par ce protocole.

Le type de traitement (encapsulation, modification...) est masqué au Message par le Protocol.
Les seules interfaces fonctionnelles exportées par un Message servent à accéder à ses données et à son état. Le cycle de vie du Message est géré par le gestionnaire de messages (Message Manager, *c.f.* 4.3.6).

4.3.3 Composants Protocol

Chaque protocole (IP, ICMP...) correspond à un composant de type Protocol. Ces composants importent et exportent une interface de communication standard, encapsulée par un seul appel : `visit()`. Lorsqu'un Message invoque l'appel `visit()` d'un Protocol, ce dernier traduit l'appel en l'un des traitements vus en 4.2 (encapsuler, décapsuler, modifier...). Ce traitement est choisi par le Protocol en fonction de l'état du Message.
Pour effectuer leurs traitements, de nombreux protocoles nécessitent une configuration. On peut citer les options IP, les paramètres IPSec, ou même la gestion de congestion TCP (TCP Reno, TCP Vegas [3]...). Par souci de flexibilité, ces configurations peuvent être modifiées selon l'application qui utilise le réseau. Elles sont donc stockées au niveau des composants Graph (*c.f.* ci-dessous).

4.3.4 Graphe de protocoles et composants Graph

Les composants Protocol exportent une interface générique, le but étant qu'ils soient interchangeables. Par conséquent, il faut poser des contraintes pour construire des graphes de protocoles cohérents. Ces contraintes sont imposées au niveau du système. Celui-ci dispose d'un graphe unique, regroupant tous les protocoles enregistrés, et indiquant tous les empilements protocolaires possibles (*c.f.* fig. 2). Ce graphe est dynamique, dans le sens où l'on peut ajouter, modifier et retirer des composants Protocol "à chaud".
Les dénominations employées (Socket, SSLSocket, DatagramSocket) sont celles de l'API de la machine virtuelle de Sun. L'API Socket est vue comme un composite, qui contient le composant de base TCP. En effet, Socket sert d'interface vis-à-vis des applications, et utilise l'implantation du protocole TCP. De même, SSLSocket est un composite qui contient Socket, car l'API SSLSocket réutilise les appels de l'API Socket. A une application, on associe un parcours du graphe 2, représenté par un composant de type Graph. Un Graph est composé de :

- un parcours unique du graphe de protocoles système ;

– pour chaque composant Protocol de ce parcours, la configuration du protocole pour l'application concernée.

Les composants Graph exportent des interfaces `getConfig()` et `setConfig()` pour pouvoir modifier la configuration d'un protocole donné pour une application donnée.

la configuration des protocoles sont modifiables. Changer le parcours du graphe de protocoles système revient à modifier l'empilement protocolaire, *e.g.* intercaler IPSec entre IP et Ethernet, ou encore remplacer TCP par UDP. Modifier la configuration d'un protocole signifie changer les options de l'en-tête IP, celles de l'en-tête TCP, *etc*.
En revanche, il faut s'assurer que ces changements de parcours ou de configuration ne cassent pas les communications en cours. Ceci est la tâche du gestionnaire de graphes.

4.3.5 Gestionnaire de graphes

Le gestionnaire de graphes gère l'association d'un composant Graph donné à une application. En effet, plusieurs applications concurrentes peuvent utiliser des empilements protocolaires différents, *e.g.* UDP / IPv6 / Eth, TLS / TCP / IPv4 / 802.11, TCP / IPv4 / IPSec / Eth...
Cet empilement est donné en partie par l'application : l'API utilisée pour accéder au protocole de Transport détermine le type de socket utilisée (TCP, datagramme, SSL, *etc*.). Les protocoles suivants sont déterminés par le système. Par exemple, la table de routage donne le lien physique à utiliser, donc le composant Protocol associé.

Le gestionnaire de graphes est aussi chargé de créer et détruire les composants Graph.

4.3.6 Gestionnaire de messages

Le gestionnaire de messages contrôle le cycle de vie des composants Message. Pour cela, les composants Message exportent les interfaces de configuration suivantes :

– `start()` : pour démarrer ou reprendre le parcours de la pile protocolaire ;
– `wait()` : pour mettre le composant Message en attente (*e.g.* s'il est placé dans une fenêtre d'émission TCP) ;
– `duplicate()` : pour dupliquer un composant Message, ses données et son état compris. Il s'agit du *design pattern* Prototype ;
– `destroy()` : pour détruire le composant.

De plus, le gestionnaire de messages instancie ces composants, lors d'un appel à l'API Socket par exemple.
Enfin, il initialise l'état des composants Message *e.g.* en indiquant leur position (le premier protocole à visiter) et leur direction (émission ou réception) dans le graphe.

Nous avons vu le rôle des composants Protocol, Message et Graph, celui des gestionnaires de messages et de graphes, ainsi que celui du graphe de protocoles système. Leur interaction est explicitée ci-dessous par un cas d'utilisation, qui fait émerger des problématiques liées aux protocoles eux-mêmes. D'autres points clés sont discutés dans le chapitre 5.

4.4 Cas d'utilisation : envoi d'un Message

A titre d'exemple, la figure 3 donne les échanges entre composants lorsqu'une application demande à envoyer des données sur le réseau. Les intitulés des appels représentés sont inspirés du langage Java. Lorsque l'application crée la socket, elle indique un certain nombre d'informations, *i.e.* le pseudo-en-tête TCP. Ce pseudo-en-tête comprend les ports TCP source et destination, ainsi que les adresses IP source et destination. On peut remarquer que cela, bien que présent dans la RFC [8] viole le principe d'encapsulation et de séparation des services en couches distinctes. Par la suite, à chaque envoi de données sur la socket, ce même pseudo-en-tête sera réutilisé. Par conséquent, il apparaît sur la figure 3 qu'à la création de la socket, un composant Message est créé pour conserver le pseudo-en-tête. Ensuite, lors de l'envoi de données sur la socket, ce composant Message est dupliqué. Son clone est mis à jour avec les données applicatives, puis démarre le parcours de son graphe de protocoles.

4.5 Fils d'exécution

La formalisation "haut niveau" (*i.e.* avec une approche fonctionnelle) d'une pile réseau ne doit pas faire oublier les problématiques élémentaires liées au système. En l'occurence, il faut choisir quels composants disposent d'un fil d'exécution (ou *thread*) et quels composants sont simplement des fournisseurs de code "statique", à l'image des méthodes de classe en Java.
Dans la pile réseau BSD, les protocoles sont des unités d'exécution et les messages sont des structures mémoire. Dans la pile *Mynus*, nous inversons les rôles : les composants Message deviennent des fils d'exécution, et les traitements des protocoles sont appelés par une API.

Toutefois, les composants Message ne sont pas les seules unités d'exécution en présence. De nombreux protocoles réseau utilisent des temporisateurs, qui, lorsqu'ils arrivent à expiration, déclenchent des traitements. Par exemple, TCP utilise un temporisateur à chaque message placé dans la fenêtre d'émission. Si le temporisateur expire, TCP réémet le paquet. Ceci implique que TCP est une unité d'exécution, capable de recevoir un signal du temporisateur.

Mynus utilise des composants Timer. Pour reprendre l'exemple de TCP, lorsqu'un message se fait traiter par le protocole (dans le sens émission), TCP le duplique (`duplicate()`) et le met en attente (`wait(`x`)`) dans sa fenêtre d'émission. Le Message déclenche un Timer de x secondes. Si le timer expire, il envoie un signal au composant qui l'a créé (le Message). Ce signal indique au Message de se dupliquer, de démarrer le clone (`start()`) et de se mettre en attente à nouveau. Si TCP reçoit un acquittement du message, il supprime celui-ci de sa fenêtre d'émission (`destroy()`), ce qui détruit également le composant Timer.

Nous avons explicité dans ce chapitre le mode de fonctionnement de *Mynus*. Les traitements des protocoles sur les messages à envoyer ou recevoir sont mis à disposition par les composants Protocol. Les composants Message sont des entités capables de suivre un graphe de protocoles (composant Graph) et de se faire traiter par chacun d'eux, de manière autonome. Les gestionnaires de messages et de graphes gèrent leurs cycles de vie et leurs configurations respectifs. L'annuaire est un élément du système, ou au choix d'un framework intermédiaire, qui permet de trouver et de lier des composants.

L'API Socket, qui apparaît sur la figure 1, relève plus de la technique d'implantation que du modèle ; elle est dont traitée dans le chapitre suivant.

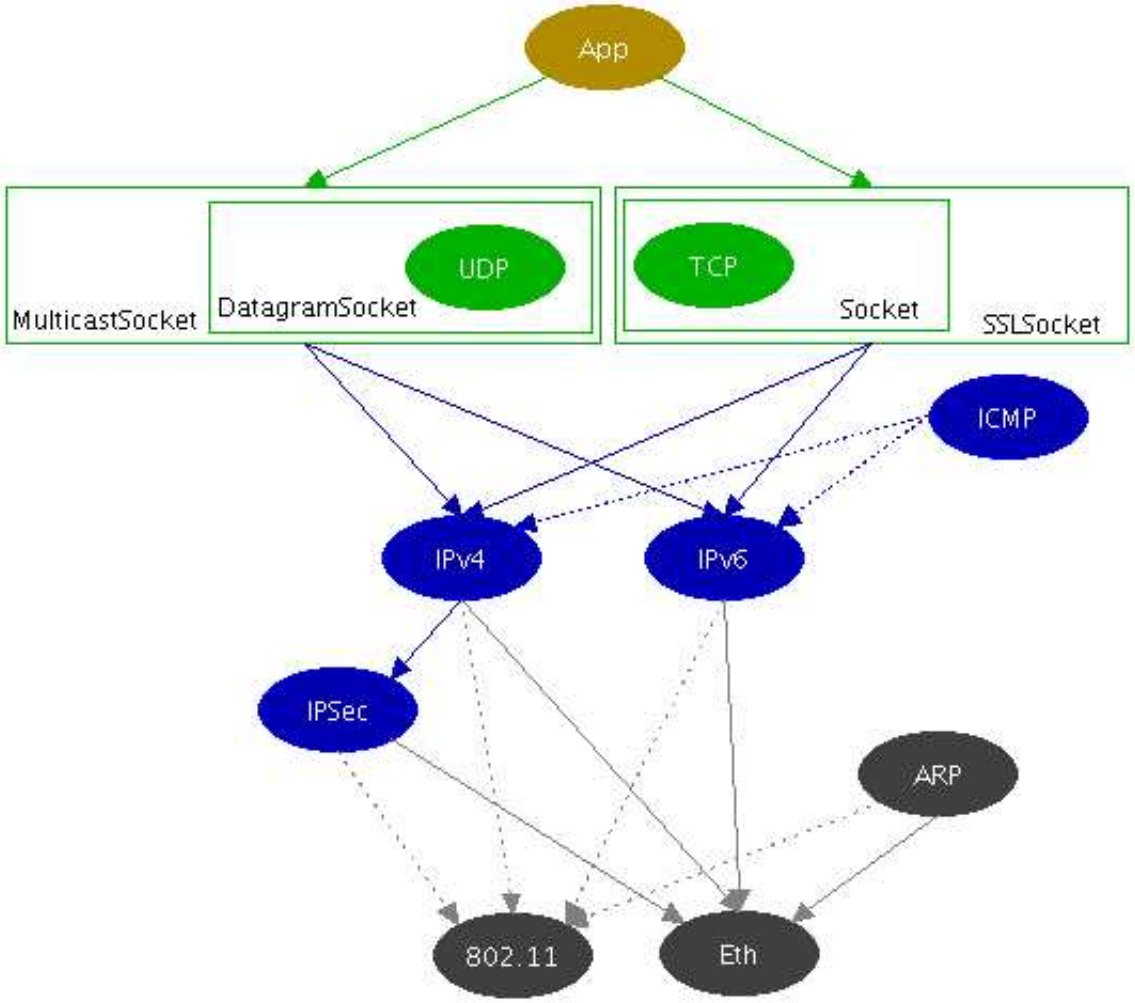


FIG. 2 – Graphe système des protocoles

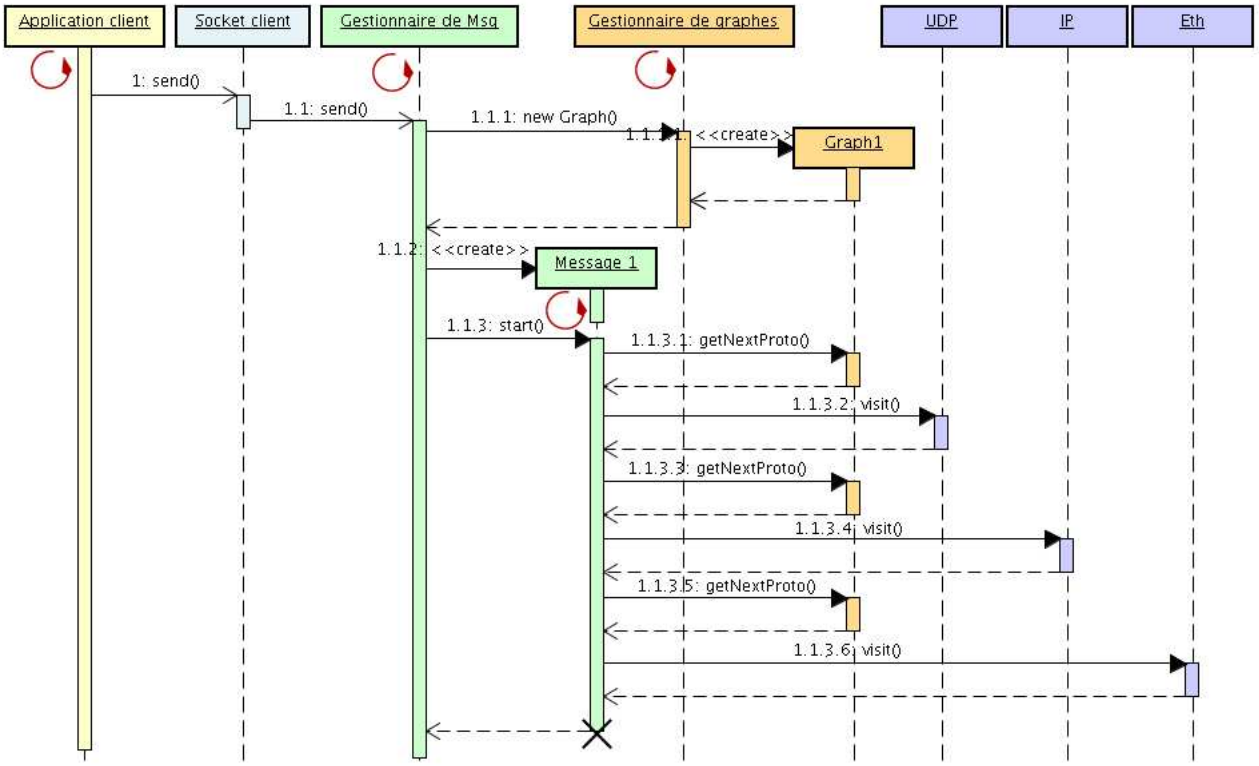


FIG. 3 – Diagramme de séquence : envoi d'un message sur le réseau par une application

5 Discussion

Une présentation générale de notre pile réseau ne permet pas d'en aborder toute le complexité. C'est pourquoi nous nous penchons à présent sur des problèmes liés aux choix d'implantation. Nous analysons également les forces et faiblesses de notre approche, ainsi que les travaux et tests à mener.

5.1 Objectifs d'implantation

Par souci de simplicité, $\mathcal{M}ynus$ est développé en Java. La pile réseau s'exécute donc dans l'espace Utilisateur, au lieu de l'espace système comme sur les systèmes d'exploitation monolithiques classiques. Nous parlons de simplicité car la notion d'interface, au coeur du modèle Fractal se concrétise trivialement en Java.
Cette implantation n'est cependant qu'à ses débuts, et se poursuit dans le cadre d'un Projet de Fin d'Etudes.

Par la suite, nous envisageons de porter $\mathcal{M}ynus$ sur une machine virtuelle à micro-noyau. Ainsi, les composants de la couche réseau seraient gérés par la JVM de la même manière que l'interpréteur, le Garbage Collector ou le ClassLoader. Cette machine virtuelle à micro-noyau peut elle-même être un système d'exploitation (*c.f.* 3.2.1). Dans ce cas, tous les services, c'est-à-dire tout ce qui est hors du micro-noyau, sont gérés de manière uniforme par le système. Tous les composants, ceux de la pile réseau compris, sont enregistrés auprès du même annuaire système, et tous utilisent le même service de liaison des composants.

5.2 Faisabilité

Pour tester $\mathcal{M}ynus$, plusieurs changements sont nécessaires, tant au niveau de la machine virtuelle Java que du système hôte. La figure 3 montre que, lors d'un envoi de message, l'API Socket utilise le Message Manager. Par conséquent, lorsqu'une application utilise l'API Socket, elle doit être redirigée, de manière transparente, vers notre implémentation de Socket plutôt que l'implémentation de l'éditeur de la JVM. Ceci se traduit par deux possibilités : ou nous modifions le code à l'intérieur de la Java API (*i.e.* le fichier *rt.jar* pour la Sun JVM), ou nous redirigeons la JVM vers notre implémentation, via une indirection ou via le Classpath.

La réception d'un message est illustrée par la figure 4. Il apparaît que, pour créer le composant Message, $\mathcal{M}ynus$ doit être averti qu'une trame Ethernet est disponible dans le *buffer* de la carte réseau. Prenons l'exemple d'un système Linux. Les protocoles "receveurs", capables de traiter une trame Ethernet, sont déclarés auprès du noyau. Le *driver* de la carte réseau est chargé d'alerter ces protocoles receveurs lorsqu'une trame est disponible.
Dans le cas de $\mathcal{M}ynus$, le gestionnaire de messages doit être enregistré comme receveur. Le *driver* notifie le Message Manager lors de l'arrivée d'une trame. Celui-ci crée alors le composant Message, initialise son état et le démarre.

5.3 Quelques points de détail

Deux des avantages d'une approche orientée composant sont la robustesse et la dynamicité.

– Robustesse : lorsqu'un service "plante", il suffit de relancer le composant correspondant. Toutefois, si ce composant était en cours de communication avec un autre composant, ce dernier est soumis à un risque potentiel de dysfonctionnement. Des solutions sont proposées, par JX par exemple (*c.f.* 3.2.1) : il s'agit d'utiliser des proxys lors des communications inter-composants, à l'image des *stubs* de RMI. Ce proxy est chargé de surveiller le bon fonctionnement de la liaison ; on peut envisager de s'en servir comme cache le temps que le composant arrêté soit relancé. Une telle solution est effectivement robuste, mais particulièrement lourde à utiliser dans une pile réseau où les échanges sont très fréquents.

– Dynamicité : un composant, dans notre cas de type Protocol ou Graph, est substituable par un autre. Ceci autorise à :

  - disposer de plusieurs versions d'un même composant, *e.g.* une version par architecture processeur, différentes implémentations d'un protocole, *etc* ;
  - mettre à jour un composant sans interrompre le service.

Pour ces deux caractéristiques, il faut noter que certains composants sont dotés d'un état. Par exemple, TCP dispose d'une fenêtre d'émission. Lorsque l'on interrompt ou substitue un composant, on peut vouloir sauvegarder son état, en vue d'une restauration future. Ceci correspond au *design pattern* Memento. Un choix s'impose alors :

- Attendre que toutes les communications avec le composant à arrêter se terminent, éventuellement envoyant des signaux à tous les composants qui lui sont liés. Ainsi, l'arrêt ne crée pas de problème d'instabilité. En revanche, il est impossible de borner le temps qu'il faudra pour arrêter le composant ;
- Arrêter brutalement le composant, pour pouvoir le remplacer immédiatement. On risque alors de casser des connexions réseau en cours, et de rendre des composants instables.

5.4 Performances et optimisations

Depuis que les systèmes d'exploitation à micro-noyau existent, ils sont quelque peu boudés par le marché. En effet, bien qu'ils soient plus robustes, leurs performances brutes chutent à cause des communications et des changements de contexte entre services. Le problème des communications est similaire avec une approche à composants. L'intérêt de $\mathcal{M}ynus$ n'est donc clairement pas de dépasser la pile réseau BSD en terme de performances, mais d'avoir une pile dynamique, adaptable, résistante aux pannes.

Il est cependant possible de réaliser certaines optimisations, afin de réduire cette perte de performances. D'un point de vue modèle, Fractal précise que toutes les fonctionnalités qui différencient un composant d'un objet sont optionnelles (introspection, cycle de vie, *etc*.). Il en est de même pour $\mathcal{M}ynus$: un composant ARP, par exemple, n'a pas besoin d'interface de configuration. Diminuer ce nombre d'interfaces rapproche le composant d'un simple objet : il s'agit d'un compromis entre performance et reconfigurabilité.

De plus, les composants sont déclarés auprès du système. Cela signifie que leur cycle de vie peut être géré de manière fine. Par exemple, lorsqu'un protocole installé sur la machine n'est pas utilisé, il n'est pas nécessaire de le garder en mémoire. Ceci permet de réduire l'empreinte mémoire de la pile.

D'un point de vue système, il peut être intéressant de limiter le nombre de composants Message, pour pouvoir borner leur temps de traitement. Si le système est muni d'un pool de composants Message, on économise le temps de création d'un composant à chaque émission / réception. De la même manière, on observe que certains protocoles envoient des messages souvent similaires. Par exemple ARP passe le plus clair de son temps à envoyer des requêtes, toujours avec le même en-tête MAC (l'adresse destination est celle de broadcast). Si le système instancie un (ou un pool de) composant-type "requête ARP", il suffit de le cloner et de renseigner les données ARP : on économise le temps de traitement du protocole Ethernet.

5.5 Exemples d'utilisation

Les possibilités de reconfiguration de $\mathcal{M}ynus$ se situent à deux niveaux. Le premier est celui des composants Graph. Il est possible d'ajouter, de supprimer et de substituer des protocoles dans la suite protocolaire utilisée. On peut, par exemple, imposer au niveau du système que toutes les communications utilisent IPSec (ou même SSL) lorsque le correspondant le permet, et de manière transparente aux applications. Inversement, on peut choisir d'utiliser UDP au lieu de TCP, toujours de manière transparente à l'application. Cela permet de décharger la mémoire en supprimant la fenêtre d'émission, et le processeur en économisant les traitements de contrôle de flux. On introduit donc des notions de sécurité et de Qualité de Service, au niveau de la pile réseau du système, et sans modifier les applications.
Les composants Graph déterminent aussi l'interface réseau à utiliser. Au lieu de se baser sur la table de routage uniquement, on peut ajouter un mécanisme de choix de la "meilleure interface". Des exemples de critères sont le débit disponible, la charge de la file d'attente, *etc*.

Le second niveau de reconfiguration se situe au niveau des composants Protocol même. Faisons de TCP un composant composite ; nous pouvons donc séparer l'encapsulation, la gestion de fenêtre d'émission, le contrôle de congestion en différents composants de base. Nous n'avons plus qu'à substituer un composant de contrôle de congestion par un autre pour passer de TCP Reno à TCP Vegas par exemple. Nous pouvons même ne mettre aucune gestion de congestion, si l'on sait que le système est confiné à un réseau local. Le traitement de TCP devient plus léger, et l'on retrouve cette notion de Qualité de Service.
Un dernier exemple est celui du routage *peer-to-peer* (*e.g.* Tapestry, CAN, Chord [1]). Le routage IP agit à l'échelle du réseau mondial, et le routage *peer-to-peer* sur une sous-partie de ce réseau. D'un point de vue service, ces deux types de routage peuvent être unifiés, par exemple pour créer des réseaux de communautés. En éclatant le composant IP en plusieurs sous-composants, nous pouvons substituer le mécanisme de routage IP par un mécanisme quelconque de routage *peer-to-peer*.

Il apparaît que cette modularité des empilements protocolaires et des implantations de protocoles permet d'utiliser, au niveau du système, des services actuellement rendus par des couches supplé-

mentaires *middleware*. Ainsi, si les performances brutes de *Mynus* sont moins bonnes que celles de la pile BSD, des tests s'imposent pour comparer notre pile réseau à composants avec l'ensemble pile réseau monolithique & logiciel *middleware*.

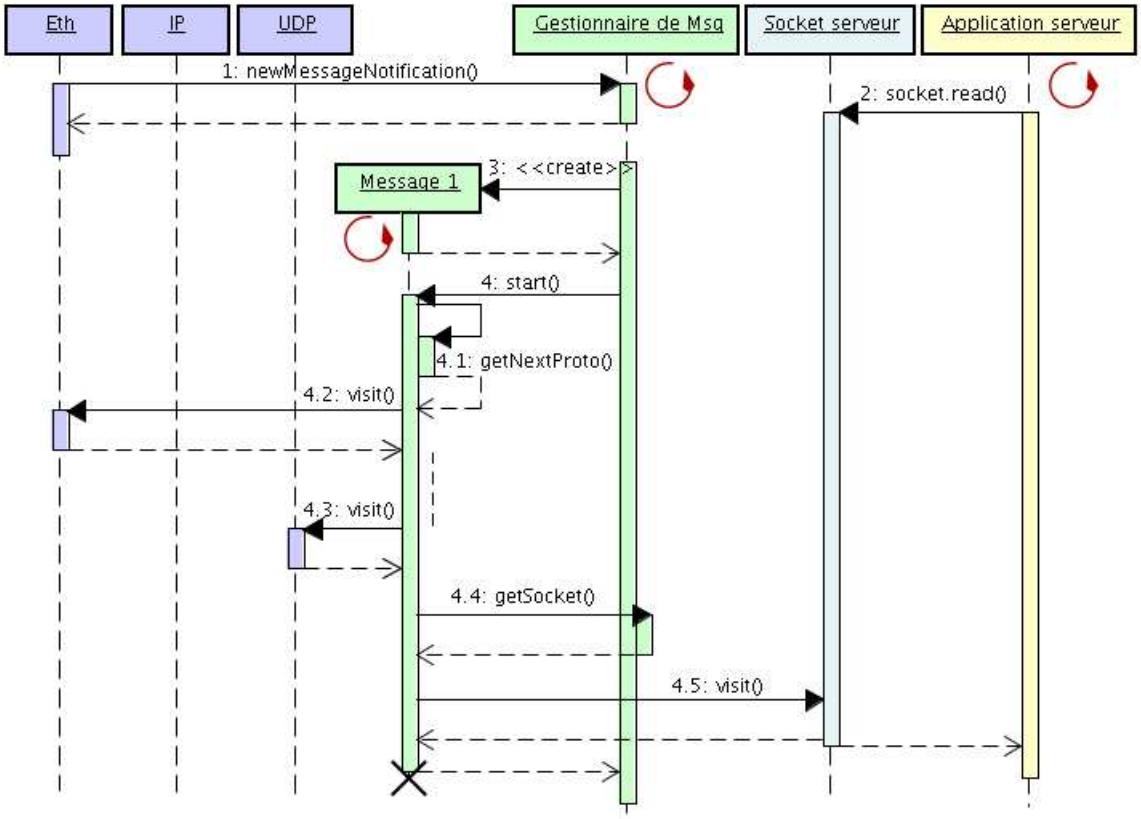


FIG. 4 – Diagramme de séquence : réception d'un message du réseau

6 Conclusion

Nous avons, dans ce mémoire, rappelé d'une part l'état de l'art de l'industrie pour les applications communiquantes, et d'autre part l'état de la recherche des approches orientées composant. Le but recherché est de croiser ces deux mondes, en modélisant $\mathcal{M}ynus$, une pile réseau fondée uniquement sur des composants.

Nous avons explicité les composants en présence, réalisant les protocoles réseau, les messages échangés, les empilements protocolaires. Nous avons ensuite discuté de points plus techniques, relatifs à l'implantation et aux techniques réseau.

Il ressort que, de par son *design*, $\mathcal{M}ynus$ est moins performant que ses consœurs monolithiques, du type BSD. Cependant, l'objectif de $\mathcal{M}ynus$ est bel et bien d'être dynamique, reconfigurable et adaptable. L'effet direct de ces propriétés est de pouvoir descendre des services, habituellement rendus par une couche *middleware*, au niveau du système. Ces services sont la gestion de la sécurité, la Qualité de Service ou le routage *peer-to-peer*. De ce point de vue, la pile $\mathcal{M}ynus$ est potentiellement plus légère qu'un ensemble pile réseau monolithique & logiciel(s) *middleware*.

Références

[1] K. Aberer and M. Hauswirth. An Overview on Peer-to-Peer Information Systems. In *Workshop on Distributed Data and Structures (WDAS-2002)*, 2002. Paris, France.

[2] The OSGi Alliance. *OSGi Service Platform*. IOS Press, 3rd edition, 2003.

[3] L. Brakmo, S. O'Malley, and L. Peterson. TCP Vegas : New Techniques for Congestion Detection and Avoidance. In *SIGCOMM*, pages 24–35, 1994.

[4] T. Braun, C. Diot, A. Hoglander, and V. Roca. An Experimental User Level Implementation of TCP. Technical Report 2650, INRIA, September 1995.

[5] E. Bruneton, T. Coupaye, and J-B. Stefani. Recursive and Dynamic Software Composition with Sharing. *Seventh International Workshop on Component-Oriented Programming (WCOP02)*, June 2002.

[6] OMNeT++ Community. OMNeT++. http ://www.omnetpp.org/.

[7] J-P. Fassino, J-B. Stefani, J. Lawall, and G. Muller. THINK : A Software Framework for Component-based Operating System Kernels. *USENIX*, 2002.

[8] Internet Engineering Task Force. RFC 793 - Transport Control Protocol, 1981.

[9] Apache Software Foundation. The Avalon Project. http ://avalon.apache.org.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[11] M. Golm, M. Felser, C. Wawersich, and J. Kleinoeder. The JX Operating System. *USENIX*, pages 45–58, June 2002.

[12] N. Hutchinson and L. Peterson. The x-Kernel : An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1) :64–76, 1991.

[13] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The Performance of Micro-Kernel-based Systems. *16th ACM Symposium on Operating Systems Principles*, October 5-8 1997. http ://os.inf.tu-dresden.de/papers_ps/sosp97.ps.gz.

[14] Sun Microsystems. Java Virtual Machine. http ://java.sun.com/.

[15] Sun Microsystems. The Java Message Service 1.0.2b Specification, October 2001. http ://java.sun.com/products/jms/.

[16] ObjectWeb. Dream : A Component-Based Communication Framework. http ://dream.objectweb.org/.

[17] ObjectWeb. The Fractal Project. http ://fractal.objectweb.org/.

[18] OPNET Technologies. OPNet Modeler. http ://www.opnet.com/products/modeler/.

[19] International Standards Organization. Reference Model of Open Systems Interconnection, August 1979.

[20] E. Prangsma. JNode. http ://jnode.sourceforge.net.

[21] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA : An Architecture for Well-Conditioned, Scalable Internet Services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001. http ://www.eecs.harvard.edu/ mdw/papers/seda-sosp01.pdf.

[22] D. Wetherall and T. Anderson. Teaching by Layers Considered Harmful for Network Education. *2nd Workshop on Networking Education, SIGCOMM*, 2003. http ://www.cs.washington.edu/homes/djw/papers/neted-curricula-7-03.pdf.

[23] G. Wright and W. Stevens. *TCP/IP Illustré, La mise en oeuvre*, volume 2. Vuibert, 1998.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifi que
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803